RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 8: Linked Lists, Basic Searching and Sorting

Lecture Slides

Dynamic Data Structures

So far, all data collections used have been either arrays or Java's built in collection classes.

To understand how such data structures and collection classes are implemented this lecture discusses the **linked list** (the simplest of self-referential data structures, with sequential access.)

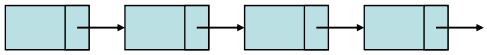
Self-Referential Structures

Many dynamic data structures are implemented through the use of a **self-referential structure**.

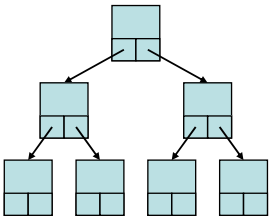
A self-referential structure is an object, one of whose elements is a reference to **another object of its own type**.

With this arrangement, it is possible to create 'chains' of data of varying forms:

DataNode
String data;
int moreData;
DataNode next;



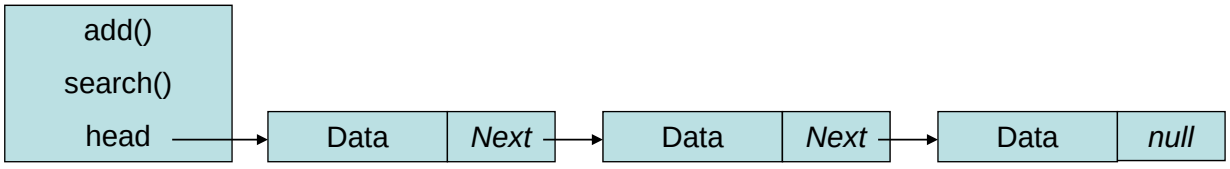
lists



trees

Linked Lists: Introduction

The simplest self-referential dynamic data structure is the **linked list**. It is simply a chain of self-referential data nodes.



The node comprises **two** items; one (or more) attributes of data, and the pointer to the next data node, commonly referred to as the **`next`** reference.

The **end of the list** is defined by a **`null`** next reference. The start (or **`head`**) of the list is contained in a **`header` class**, which also encapsulates all the list's functionality. This helps the linked list become an independent, cohesive package. The header class provides the API to access the list.

Linked Lists: the `List` Node

The list node is a simple self-referential structure that stores an item of data, and a reference to the next item.

```
class ListNode {

  private int data;

  ListNode next;


  public ListNode(int data){
    this.data = data;
    this.next = null;
  }
}
```

The data variable is where the information to be stored resides. It may be of any primitive or reference type but is usually generic (java.lang.Object or parameterised type <T>).

The next variable is the self-referential link to the next data item.

The constructor initialises the node object by storing the data that was given as an argument, and sets the next reference to **`null`**.

Linked Lists: the `header` Class

The **`header`** class is the public interface for the linked list. It is where the functionality is stored (as methods), and contains a link to the first item of the list (the 'head'.)

```
class List {

  private ListNode head;


  List() {
    head = null;
  }


  add();    // discussed later
  find();   // discussed later
  delete(); // discussed later
}
```

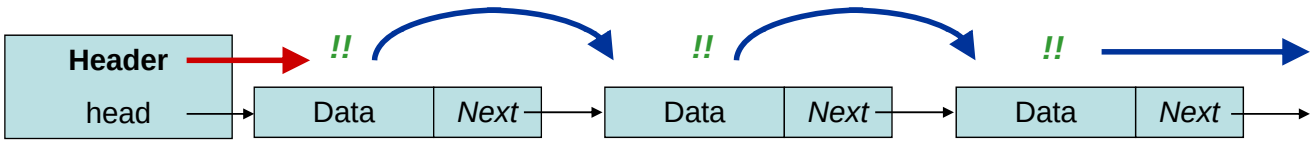
The `head` variable is a reference to the first item in the list.

The constructor initialises the list by setting the head to `null` (an empty list.)

The methods provide a way to use the list. They each access the structure through the head reference.

Linked Lists: List Traversal (1)

It is sometimes necessary to traverse the entire length of the list to perform some function (for example, to count the number of items, or display summary information.)



Step 1: Step through the list from the header node forward.

Step 2: Perform the desired operation at that node.

Step 3: Move onto the next node, until the end of the list is reached.

List traversal forms the basis of many of the list manipulation operations such as add, retrieve and delete.

Linked Lists: List Traversal (2)

The code below will traverse the entire list, and print out the data contained in each node.

```
public void traverse() {
    ListNode current = head;

    while (current != null) {

        System.out.println(current.data);

        current = current.next;
    }
}
```

Step 1: Maintain a variable to store the current position in the list.

Step 2: Continue stepping through the list, until the end of the list (a **`null`** reference) is reached.

Step 3: At each step, print out the data present in the current node.

Because of the way a single link list is defined, we can only access data in one direction, and sequentially (one item after another.)

Linked Lists: Adding Data

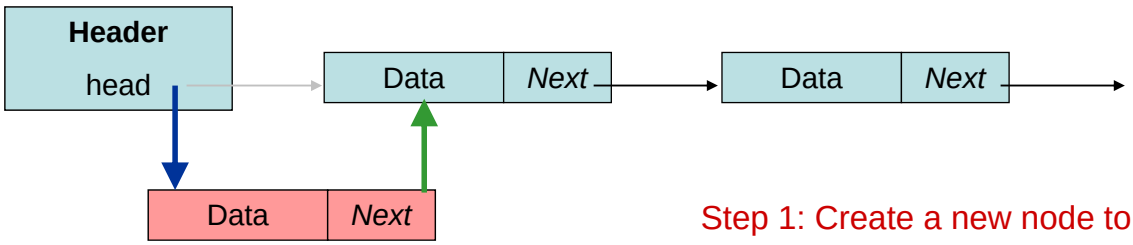
Data is added to a linked list by wrapping the data to add into a node, and then placing that node at the appropriate place in the data structure. Note that the API of the list would usually hide the ‘wrapping’.

Depending on the circumstances and purpose of the list, there are a number of places where data may be added:

- **At the start (head) of a list**
- **In the middle of the list (e.g. at a specific index or node count)**
- **At the end (tail) of the list**
- **At the appropriate place to preserve sort order**

Linked Lists: Adding Data to the Head (1)

Adding data to the head of a list is the easiest and quickest way in which it can be done.



Step 1: Create a new node to store the given data.

Step 2: Set the new node’s next reference to the first node.

Step 3: Reset the head reference to point to the newly created node.

The order in which the link manipulations are done are very important; they must always be done from right to left, otherwise data nodes will be lost.

Could use temporary variables and do in any order but is less elegant.

Linked Lists: Adding Data to the Head (2)

Here is the code for the previous slide:

```
public void addToHead(ListNode aNode) {

  aNode.next = head;

  head = aNode;
}
```

And you would use it thus:

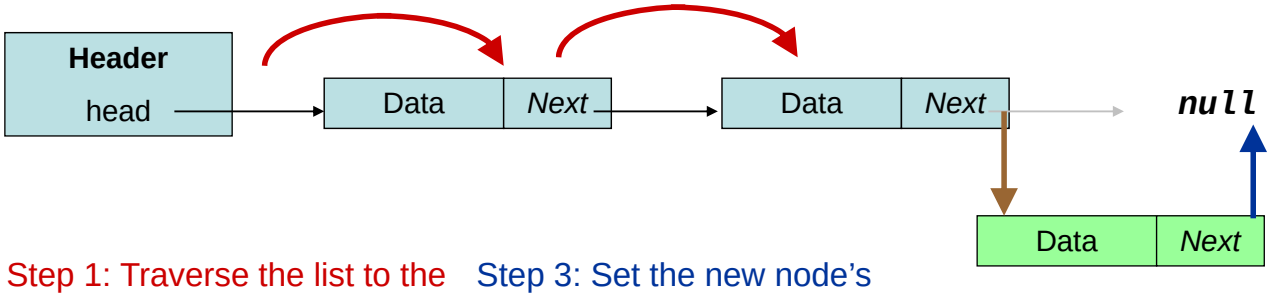
```
ListNode newNode = new ListNode(data);  // call constructor

addToHead(newNode);                     // call method
```

Question: Does the caller really need to know about the ListNode class?

Linked Lists: Adding Data to the Middle or Tail (1)

Adding data to the middle (e.g. at a specific index or node count) or tail of the list is essentially the same process. The diagram below illustrates adding to the end (tail.)



Step 1: Traverse the list to the desired insertion point (in this case, the last item of the list.)

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

Step 4: Reset the insertion point's next reference to point to the newly created node.

Linked Lists: Adding Data to the Middle or Tail (2)

```
public void addToTail(ListNode aNode) {

  ListNode insert = head;
  // check for empty list
  if (head == null)
    head = aNode;
  else
  {
    // could instead check for index here if
    // inserting in middle of list
    // while we are not at the end of the list
    while (insert.next != null)
       insert = insert.next;

    aNode.next = null;
    insert.next = aNode;
  }
}
```

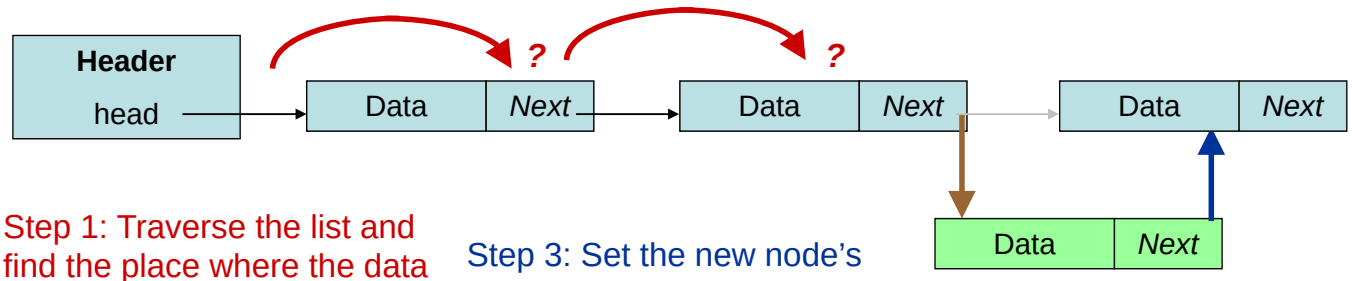
And you use it:

```
ListNode newNode = new ListNode(data);

addToTail(newNode);
```


Linked Lists: Adding Data in Sort Order (1)

Adding data in sort order is somewhat more complex. This is because we do not know beforehand which insertion strategy (head, middle, tail) is needed, and must be determined on-the-fly.



Step 1: Traverse the list and find the place where the data should be stored. Both 'current' and 'previous' references will need to be maintained.

Step 2: Create a new node to store the given data.

Step 3: Set the new node's next reference to that of the insertion point node.

Step 4: Reset the insertion point's next reference to point to the newly created node.

Note: Steps 3 and 4 will need to be implemented differently if the insertion point is the head.

Linked Lists: Adding Data in Sort Order (2)

```
public void addInOrder(ListNode aNode) {
  ListNode current = head;
  ListNode previous = null;
   while ((current != null) && (current.data <= aNode.data)) {
    previous = current;
    current = current.next;
  }
  // head
  if (current == head) {
    aNode.next = head;
    head = newNode;
  }
  // tail
  else if (current == null) {
    previous.next = aNode;
  }
  // middle
  else {
    previous.next = aNode;
    aNode.next = current;
  }
}
```


Linked Lists: Adding Data in Sort Order (3)

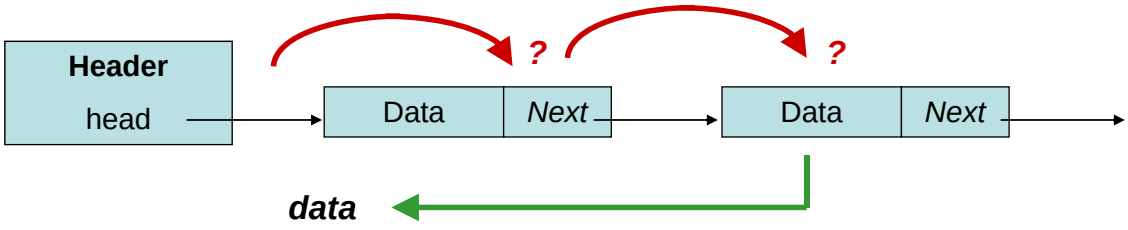
And you use it:

```
ListNode newNode = new ListNode(data);

addInOrder(newNode);
```

Linked Lists: Retrieving Data (1)

Data retrieval consists of traversing the data structure until a requested node is found (e.g. a specific value or an item at an index). If the data is not found, some form of failure signal should be returned instead (e.g. boolean return value).



Step 1: Traverse the list.

Step 2: While traversing, compare the search 'key' value with the data in each node.

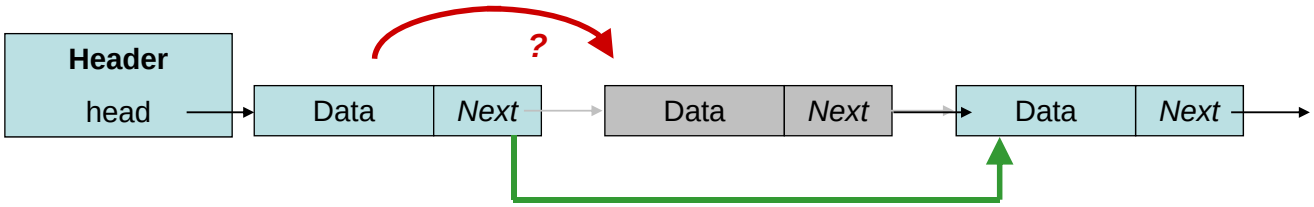
If the data keys match, return the data portion of the node.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Linked Lists: Deleting Data (1)

Deletion is very similar to retrieval. As before, the list is traversed to find data matching a given 'key' value. However, instead of returning the data, the node is to be deleted.

The node can be deleted by having the next references 'jump over' the node to delete. To do this, the node before the one to delete must be known, and as such, the traversal needs to keep track of two references.



Step 1: Traverse the list, maintaining both current and previous references.

Step 2: If the search key matches the current data node, 'jump over' the current node, and return success.

Step 3: If the list is exhausted without the data being found, return a search failure signal.

Data Storage, Retrieval and Sorting

One of the primary applications of computers is the **storage** (through data structures) **and retrieval** (via **search** algorithms for those structures) **of data**.

Unorganised data is easier to manage (since data additions and deletions can be done easily) but slower to **search** (since entire collection must be searched using **linear search**)

Therefore where data retrieval speed is important, data should be sorted or indexed (e.g. hashing) in some way.

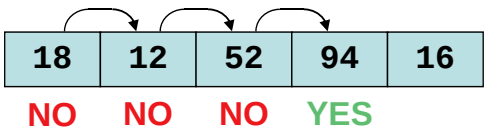
Common sorting algorithms include quick sort, merge sort, insertion sort, selection sort and bubble sort. Large data sets are often sorted on disk rather than in memory using techniques such as b-tree (not covered in this course).

Selection sort is covered in this lecture to provide a simple example.

The Linear Search

The **linear search** algorithm is the simplest of all searches. It merely **iterates through the collection**, checking for a match between the data at that position and the key. Only useful for unsorted/non-indexed data (use Binary Search otherwise).

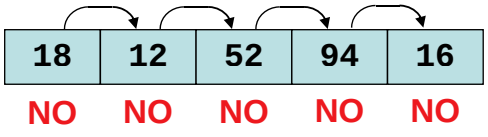
*(looking for **94**)*



Result: *index 3*

If the data is **not present** in the collection, the linear search will look through **all elements**, and **finish with no result.**

*(looking for **78**)*



Result: *failure*

The Linear Search Algorithm

The **linear search algorithm** psuedocode is simply:

```
loop i from 0 to the last element
   if collection[i] is equal to the target
      return i
   end if
end loop
return -1
```

The basic algorithm is similar for a **linked list**. However, instead of looping through element indices, the loop will continue moving through the list nodes **until the end of the list has been reached.**

The Big-O Notation

A common way of assessing algorithm complexity is **Big-O** notation. This is an evaluation of how running time increases as the volume of data (N) increases.

A crude Linear Search may examine *every* item in the collection. Therefore, running time has a direct linear relationship to volume of data. We call this a performance of **O(N)**, where N is the volume of data. This means that the complexity of the search is linearly related to the size of the data. Examples of Big-O(N): N, 3N, 1/2 N, N + 3, but not N^2, N^3, Log(N), 10^N

Big-O means that the complexity *is an approximation of the specified N*

A Linear Search should stop when the correct key is found.

- ***Best*** case is when the key is in the first position. A performance where only one operation is needed is **O(1)**.
- ***Worst*** case is when the key is in the last position and all items must be examined: **O(N)**.
- ***Average*** case is when the key is about halfway, giving a performance of **O(N/2)**.

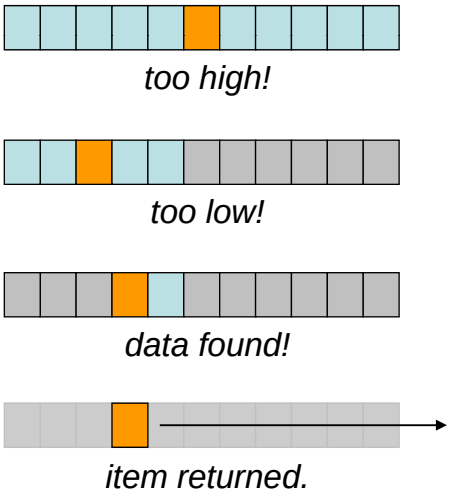
Java API Arrays.sort() algorithm uses Merge Sort which has **O(N * log(N))** performance

The Binary Search

The **binary search** algorithm is far more efficient **O(log N)** than the linear search. However, **it relies on two conditions**: the data that is being searched **sorted and is indexed**.

It works by looking at the **middle element of the collection**; depending on whether that item is **larger** or **smaller than the data searched for**, it **disregards the other half** of the collection.

This is **repeated** until the target data has been **tracked down**, or the **boundaries are invalid** (the data does **not exist**.)



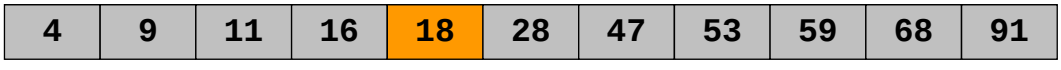
The Binary Search: An Example

Consider the collection below, searching for **18**:

***28** is too **high** – the **right hand** half is discarded.*

***11** is too **low** – the **left hand** half is discarded.*

***16** is too **low** – the **left hand** half is discarded.*



*The target (**18**) has been **found**.*

The Binary Search: Another Example

Consider the collection below, searching for **37**:

***28** is too **low** – the **left hand** half is discarded.*

***59** is too **high** – the **right hand** half is discarded.*

***47** is too **high** – the **right hand** half is discarded.*

*There is **no more data** – therefore, the target is **not present**.*

The Binary Search Source Code

```
public static int binarySearch(int[] numbers, int target)
{
   int index, left = 0, right = numbers.length - 1;

   while (left <= right)
   {
      index = (left + right) / 2;

      if (target == numbers[index])
         return index;

      if (target > numbers[index])
         left = index + 1;
      else
         right = index - 1;
   }

   return -1;
}
```

Ensure the boundaries are valid.

Find the mid-point.

If a match is found, return the position.

Otherwise, reset the boundaries to ignore the irrelevant half.

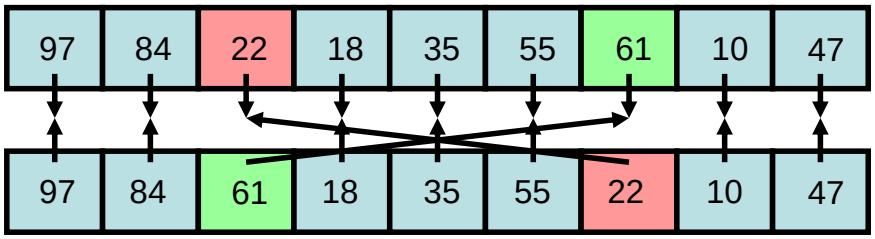
Sorting Algorithms

The performance of a sorting algorithm is affected by:

1. machine time needed for running the coded algorithm
2. memory requirements (due to recursion, see topic 9)

(2) is not always as critical since most searching algorithms have a memory requirement directly related to the size of data. (If records are large, they can be sorted *indirectly* by sorting the keys and references.)

When a key is moved, so is the reference to the rest of the record, which stays as is.



** (1) and (2) are interrelated. In general, the more complex sorting algorithms need less machine time. If sorting is needed infrequently, or there is only a small amount of data to be sorted, it is better to use a simple method. For small data sets, they are sometimes more efficient.

Selection Sort (1)

The **selection sort** algorithm gets its name, because it works on the **successive selection of items** in a particular order.

Consider a selection sort algorithm to sort a collection of integers in smallest to largest order.

The first step is to find the **smallest piece of data** in the collection, and **swap it into the first** position in the collection .

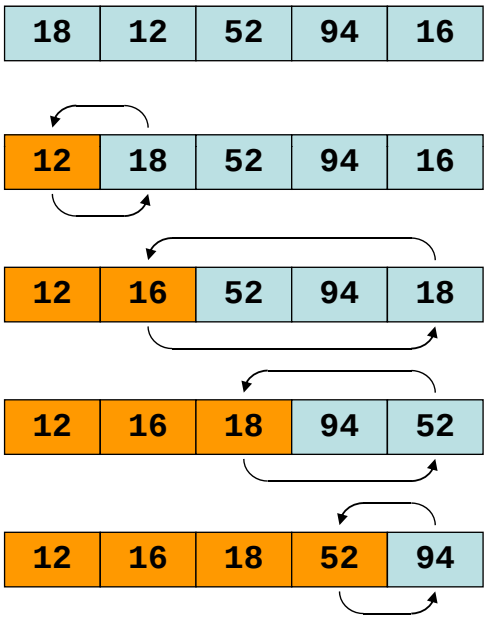
It will then find the smallest data present **in the remaining section** of the collection, and swap it with the second position.

This process continues, finding the smallest data in the (shrinking) subsection of unprocessed data, until there is **only one data item left.**

Best case is when the data are already sorted; we can avoid unnecessary exchanges (as with bubble, so performance is **O(N)**. The worst case, data are in inverse order, still has a performance no greater than **$O(N^2)$**.

The Selection Sort (2)

Consider the collection below:



This is the collection, before the selection sort commences.

It finds **12** as the smallest number, and swaps with **index 0**.

It finds **16** as the smallest number, and swaps with **index 1**.

It finds **18** as the smallest number, and swaps with **index 2**.

It finds **52** as the smallest number, and swaps with **index 3. The collection is now sorted.**

Selection Sort: Source Code

```
public static void selectionSort(int[] numbers)
{
   int min, temp;

   for (int i = 0; i < numbers.length-1; i++)
   {
      min = i;
      for (int scan = i+1; scan < numbers.length; scan++)
         if (numbers[scan] < numbers[min])
            min = scan;

      // swap the values
      temp = numbers[min];
      numbers[min] = numbers[i];
      numbers[i] = temp;
   }
}
```